VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Umělá inteligence pro tahovou RPG hru

Artificial Inteligence for a RPG Game

2014 Robert Bílý

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání bakalářské práce

Student: **Robert Bílý**

Studijní program: B2647 Informační a komunikační technologie

Studijní obor: 2612R025 Informatika a výpočetní technika

Téma: Umělá inteligence pro tahovou RPG hru
Artificial Inteligence for a RPG Game

Zásady pro vypracování:

Umělá inteligence je velice rozsáhlý obor. Má uplatnění převážně v robotice, avšak nejznámější aplikace jsou v počítačových hrách. Cílem této bakalářské práce je naprogramovat libovolnou a jednoduchou tahovou RPG hru a vytvořit pro ní umělou inteligenci. Hlavní pozornost bude student věnovat algoritmům pro Pathfinding.

Student provede tyto kroky:
1. Naprogramuje tahovou počítačovou hru žánru RPG.
2. Vytvoří přívětivé API pro připojení modulu umělé inteligence.
3. Prozkoumá možnosti a techniky umělé inteligence.
4. Implementuje modul umělé inteligence.

Seznam doporučené odborné literatury:

Millington, Ian and Funge, John, "Artificial Intelligence for Games", Second Edition, 2009 ISBN 0123747317, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA

Formální náležitosti a rozsah bakalářské práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí bakalářské práce: **Ing. Karel Mozdřeň**

Datum zadání: 01.09.2013
Datum odevzdání: 07.05.2014

doc. Dr. Ing. Eduard Sojka
vedoucí katedry

prof. RNDr. Václav Snášel, CSc.
děkan fakulty

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 7. května 2014 .....

Rád bych poděkoval vedoucímu bakalářské práce Ing. Karlu Mozdřeňovi za odbornou pomoc a konzultaci při vytváření této práce.

Abstrakt

Umělá inteligence je stále se rozšiřujícím oborem, nejen v herním průmyslu. Ve hrách je inteligence využívána k vytvoření počítačem ovládaných hráčů, tzv. inteligentních agentů. Cílem této bakalářské práce je naprogramovat jednoduchou tahovou strategii obsahující agenty schopné nahradit lidského hráče. Hlavní pozornost je věnována algoritmům pro vyhledávání nejkratší cesty, které jsou používány pro pohyb jednotek. Dále jsou implementovány i algoritmy pro efektivní rozhodování hráče.

Klíčová slova: umělá inteligence, tahová strategie

Abstract

Artificial intelligence is rapidly growing, not only in gaming industry. In games intelligence is used to create computer-controlled players, so-called intelligent agents. Objective of this bachelor thesis is to program a simple turn-based strategy containings agents able to replace human player. Attention is mainly focused on algorithms for searching shortest path for units' movement. There are also implemented algorithms for efective player's decisions.

Keywords: artificial intelligence, turn based strategy

Obsah

1 Úvod

Cílem této bakalářské práce je navrhnout a implementovat umělou inteligenci pro tahovou strategii. Umělá inteligence se bude skládat ze základních akcí, které jsou potřebné pro chod samotné hry. Navíc je hlavní pozornost věnována algoritmům pro vyhledávání nejkratší cesty k cíli. Tyto algoritmy, které se souhrnně nazývají algoritmy pro Pathfinding, pomohou zlepšit efektivitu a obtížnost inteligentního agenta (počítačem ovládaného hráče).

Práce je rozdělena do několika částí. Každou část je nutné splnit, abychom mohli splnit i část následující. Jelikož je závěrečnou části již zmiňovaný Pathfinding, je třeba nejdříve projít implementací základních prvků hry, abychom mohli tuto část realizovat. Základním stavebním kamenem bude herní prostředí, které je popsáno v následující kapitole. Prostředí bude využíváno jednotkami, které se v něm budou pohybovat. Tyto jednotky budou řízeny hráči, jak lidskými tak i těmi ovládanými počítačem. V první části se budu zabývat jen lidským hráčem, který je jednodušší na implementaci, a lepší pro zkoušení herních mechanismů. Až bude hráč hotov, budu moct pokračovat částí druhou.

Druhá část se zabývá připojením modulu umělé inteligence. V této fázi je herní prostředí hotové, a implementován lidský hráč. Nyní je tedy nutné vytvořit inteligentního agenta, který hráče dokáže nahradit. Tímto krokem se pořád ještě nedostáváme k samotné implementaci Pathfindingu. Nejprve je nutné vytvořit agentovi základní pravidla, kterými se bude řídit. Analýza momentálního rozložení hrací desky, následný pohyb a útok. To vše by měl agent v této fázi zvládnout.

Posledním krokem je implementace vyhledávání nejkratší cesty k cíli. Pohyb jednotek tímto přestává být nahodilý, každá jednotka bude schopna najít tu nejkratší cestu, po které se vydá. O Pathfindingu se budu více rozepisovat v kapitole 4.

1.1 Umělá inteligence

Umělá inteligence je věda, která se snaží porozumět inteligentním projevům zejména lidské mysli, a promítnout tyto projevy do neživých věcí, zpravidla počítačů. Inteligentními projevy rozumíme učení, řešení problémů, uvažování a podobně. Umělá inteligence tedy neslouží k ničemu jinému než aby dodala neživému organismu možnost přemýšlet a zvládat každodenní myšlenkové pochody tak jako člověk nebo třeba zvíře. V dnešní době je tato technologie už tak pokročilá, že existují odvětví, ve kterých uměle vytvoření roboti dokáží zvládat procesy dokonce lépe než člověk. Jedno z těchto odvětví jsou i hry, kde vytvořit důstojného protivníka je velice důležité. Přesto se najdou činnosti, ve kterých nejsou počítače zdaleka tak výkonné jako lidský mozek. Mnohdy jsou to činnosti, které my shledáváme triviálními, například rozeznávání obličejů. Pak jsou zde i myšlenkové procesy, ve kterých ani nejde předpokládat, že by mohla uměle vytvořená bytost vynikat, například kreativita. A to je důvod, proč je vývoj umělé inteligence stále tak důležitý. Je třeba zamyslet se nad těmito nedostatky, a vyvinout přístroj, který se i v základních činnostech bude moct vyrovnat člověku. Pokud se tedy nebojíte, že by pak roboti ovládli Zemi a udělali z nás otroky.

Tato věda je velice úzce spjata se sociálními vědami jako jsou psychologie nebo filozofie. Asi je zřejmé, proč tomu tak je. Nejdříve je důležité pochopit, proč se člověk chová tak jak se chová, a pochopit podstatu toho, co nás dělá inteligentními a správně se rozhodujícími organismy. Pochopit naše mentální procesy.

Umělá inteligence se promítá také do techniky. kde jsou vyvíjeny algoritmy umožňující provádět lidské úkony. A právě toto odvětví je pro tuto práci nejdůležitější. Pochopit myšlení hráče, navrhnout ho, a vytvořit inteligentního agenta, který dokáže toto chování napodobit, vyrovnat se mu, a v nejlepším případě ho i překonat.

1.2 Umělá inteligence ve hrách

Umělá inteligence je také využívána v herním průmyslu, který je v dnešní době velice rozšířený. Je zde tvořena algoritmy určujícími předdefinované chování a reakce v prostředí kontrolovaném programátorem, a obsahuje všechny instrukce, které jsou nutné k simulaci lidského chování. Tato inteligence je nedílnou součástí her, a z velké části určuje jejich kvalitu.

Od svých začátků prošla herní umělá inteligence velikým pokrokem. První videohry inteligenci vůbec neobsahovaly. Byly většinou tvořeny pro dva lidské hráče. Ale to se brzy změnilo. S titulem *Invaders* přišly první známky inteligentních agentů, kteří se pohybovali ve formacích, stříleli náhodně, a podle úrovně se jim zvyšovaly schopnosti. *Galaxian* již obsahovaly mnohem sofistikovanější a flexibilnější vzorce pohybů. Jednotky se také pohybovaly ve formacích, ale navíc dokázaly manévrovat a utrhnout se z formace, pokud to bylo nutné. Poté přišel velký hit, a jedna z nejznámějších her vůbec. *Pac-Man*, vykrojený kruh, který bavil děti i dospělé po celém světě. A zaslouženě. *Pac-Man* přišel s mnohem komplexnější umělou inteligencí, než nabízely konkurenční tituly. Pac-Manovými protivníky byli čtyři duchové, kde každý z nich byl, co se týče chování, jedinečný. A tím to ještě nekončilo. S čím dál efektivnější inteligencí začali být hráči náročnější, a vyžadovali mnohem více než doposud. Umělá inteligence se pomalu začala stávat hlavním lákadlem her, a na její vývoj bylo vynaloženo stále více úsilí. Úsilí, které dovolilo vzniknout mnoha skvělým titulům. *Metal Gear*, střílečka, která byla tak dobrá, že si vysloužila mnoho pokračování a pořádnou fanouškovskou základnu. V této hře se nepřátele dokázali rozhodovat na základě výstřelů. Při spuštění alarmu vám hned bylo jasné, že je budete mít za zády. Tak dobře bylo jejich chování implementováno. Co přišlo pak? Co jiného, než *Metal Gear 2: Solid Snake!* Pohyby hlavou, detekce kroků, zlepšené vyhledávání nepřátel, to vše už bylo samozřejmostí. Hra na konzoli Nintendo 64, *GoldenEye 007*, je dalším krásným příkladem vývoje umělé inteligence. Při střelbě se NPC (počítačem ovládaná postava) dokáže skrčit nebo si i lehnout, aby zaujal lepší postavení, a vyhýbal se střelám. Jelikož jim byla vždy známa pozice hráče, byli vytvořeni tak, aby si ho nevšímali, dokud hráč nepřekročil určitou hranici. Takovéto podvádění bylo sice docela nefér, ale v té době bylo běžné tvořit hry tímto způsobem. Je holt těžké vyrovnat se lidskému hráči, a tak musí programátoři vytvářet kličky a výhody pro své agenty, aby jim dali určitou výhodu. O tomto způsobu vytváření inteligentních agentů se více dočtete v následující podkapitole. A teď zpátky k *Metal Gear*. V *Metal Gear Solid 2: Sons of Liberty* se poprvé objevila týmová práce. Nepřátelé pracovali společně a dorozumívali se, aby pokořili hlavní postavu. Dokonce i vyjadřování

a gestikulace pokročila obrovským vývojem. Agenti v této hře projevovali emoce - strach, když například hleděli do hlavně hráčovy zbraně.

1.2.1 Jednotlivé techniky herního AI

Skriptování - Skriptování, nebo také umělá inteligence řízená pravidly, je základní typ techniky používané ve hrách. Zakládá se na jednoduchém způsobu, kdy programátor píše instrukce ve stylu „Když se stane událost, proveď akci“. Pokud to aplikuji na situaci v minulé podkapitole, platilo by, že pokud se spustí alarm, agent začne prohledávat budovu, a hledat vetřelce.

Náhodné skriptování - Obyčejné skriptování s náhodnými akcemi agenta. Tento způsob dodává chování větší nevyzpytatelnost. Agent už nereaguje na podněty jedním určitým způsobem, ale je mu dodána možnost náhodně se rozhodnout (svobodná vůle), a vykonat jednu z nabízených akcí. Tento způsob je stejně jednoduchý jako klasické skriptování, ale efektivita je mnohem vyšší. Hráč najednou nemůže jednoduše předvídat agentovy pohyby, a musí se přizpůsobovat jeho reakcím. Jako příklad pozměním příklad minulý. Pokud se spustí alarm, agent začne prohledávat budovu, a hledat vetřelce, nebo zavolá posily, nebo zůstane na místě, a bude střežit místnost.

Skriptování na základě charakteru agenta - Předchozí skriptování mělo navíc náhodné chování pro každého agenta, což přidávalo větší množství možností a scénářů, které mohly nastat. Toto skriptování dodává možností ještě více. Skriptování na základě charakteru implementuje agentům rozdílný způsob chování. Každý agent se tedy bude chovat jinak, a je už jen na autorovi, zda tyto agenty rozliší i vizuálně, nebo ne. Pokud by to neudělal, byla by obtížnost zase o něco vyšší, protože by tím hráče zmátl. Každý agent má svou vlastní tabulku chování, která ho činí jedinečným a méně předvídatelným.

Pathfinding - Algoritmus sloužící pro vyhledání cesty z bodu A do bodu B. Pathfinding funguje tak, že zná polohu počátečního bodu, a snaží se najít nejkratší nebo nejjednodušší cestu z tohoto bodu do bodu cílového. Programátor musí mít také na paměti mnoho faktorů ovlivňujících tuto cestu. Například kde se vyskytuje protihráč, zda se někde bojuje a nebo zda by měl agent při pohybu vykonávat i jiné činnosti. Ve 3D světě, kde se objevují různorodé terény a překážky, je někdy problém vyvinout dostatečně efektivní pathfinding, který by činil agenta schopným se libovolně pohybovat po celé mapě a zachovat si přitom realistické chování.

Rubber Banding [2] - Toto slovní spojení pro jistotu nepřekládám, protože jsem nikde nenašel jeho český ekvivalent, ale nazval bych to jako „podvádění“. Algoritmus spočívá v tom, že dává umělé inteligenci výhodu v podobě schopností, které lidský hráč nemá. Agent v podstatě podvádí, aby se hráči vyrovnal. Příkladem by mohla být závodní hra, ve které získá soupeřovo auto vyšší rychlost, když se vzdálí z dohledu hráče. Tato rychlost by klidně mohla odpovídat i dvojnásobku maximální rychlosti hráče. Důležité je, aby tuto zradu nemohl hráč zaregistrovat. Pokud soupeř znovu hráče dojede, jeho rychlost a schopnosti jsou opět sníženy na běžnou úroveň.

1.3 Tahová strategie

Tento herní žánr preferuje pomalejší přístup k provádění akcí. Zatímco strategie probíhající v reálném čase jsou uspěchanější a chaotičtější, tahové strategie jsou pro hráče, kteří chtějí mít čas k uvažování nad příštími tahy. Každá jednotka může během jednoho tahu provést určitý počet akcí. Poté, co hráč provede všechny zamyšlené akce, předá „žezlo“ druhému hráči tím, že ukončí tah. Poté je na řadě druhý hráč, který provede své akce. Můžeme si to představit jako šachy, kde má hráč možnost pohnout jednou figurkou, a poté předá tah protihráči, který také pohne jednou figurkou. Díky tomuto přístupu je jednodušší vymýšlet a taktizovat, protože hráče netlačí čas, a může si vše dobře promyslet. Vznikají tak velice komplexní strategie, které by v reálném čase nemohli hráči zvládat, a vůbec by nevyužili herní potenciál. Tahová strategie nemusí být přímo celá hra, ale jen její součást. Existuje spousta her, kde je hra hrána v reálném čase, a když dojde na souboj nebo jinou událost, hra se přepne do režimu tahové strategie. Například série *Final Fantasy* využívá těchto prvků. Pokud jde čistě o tahovou strategii, jejím nejznámějším zástupcem je série *Heroes of Might and Magic*. Jeden tah zde představuje jeden uplynulý den. Na začátku každého tahu získá hráč suroviny potřebné ke stavbě budov a najímání jednotek. Hráči mají minimálně jednoho hrdinu, se kterým se mohou během jednoho tahu přesunout o určitý počet bodů pohybu. Během tohoto tahu je hrdina schopen i útočit na neutrální příšery nebo soupeřovy jednotky. Boje se také odehrávají v tazích, kde se jednotky v útocích střídají.

2 Herní mechanismy

Mnou vytvořená tahová strategie je inspirována fantasy hrou *Ancient Empires*. Hra byla vytvořena společností Macrospace pro mobilní telefony v roce 2005. Herní mechanismy proto budou víceméně totožné s mechanismy této hry. V mé hře jsem aplikoval i vlastní nápady, které mě napadly už před devíti lety, kdy jsem si hru poprvé zahrál. Všechny jsou ale jen vedlejší, a nijak převážně nemění chod hry.

Hra se odehrává ve 2D prostředí fantasy světa, tvořeného rozmanitými stavbami a terény, kde na jedné straně stojí vojsko Aliance a na druhé vojsko Hordy. Hráči ovládají jednotky své frakce, a snaží se v bitvách zničit jednotky soupeřovy. Každý hráč na začátku svého tahu získá příděl zlata, které slouží k nákupu jednotek, kterých si hráč může koupit libovolný počet. Samozřejmě na to musí mít dostatek peněz. V každém tahu je jednotka schopna provést jeden pohyb, a poté zaútočit. Bitvy se týkají dvou jednotek útočících na sebe, kde první jednotka zaútočí, a druhá útok vrací, pokud jí zbývají síly. Cílem těchto bitev je zničit soupeři všechny jeho jednotky, a mít tak volnou cestu k jeho hradu, což je v podstatě srdce hráče. Když je hrad zničen, hra končí, a hráč prohrává.

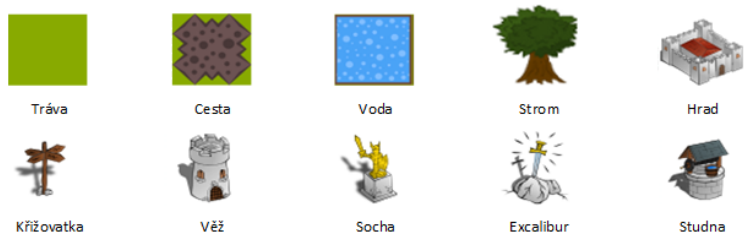
2.1 Prostředí

Prostředí je tvořeno políčky, kde každé pole reprezentuje určitý terén. Terén ovlivňuje cenu pole, a také zda se vůbec na toto pole může jednotka přesunout. Cena pole je velice důležitá pro pohyb jednotek, a odráží se na celém algoritmu pro vyhledávání cesty. Čím větší je tato hodnota, tím více bodů pohybu musí jednotka utratit, aby se na toto pole mohla přesunout. Více o ceně polí se rozepíši v kapitole 4.1. Kromě terénu může pole obsahovat také stavbu. Většina staveb má jen kosmetický účel, aby hra nevypadala moc obyčejně. Jediná stavba, která je naopak nesmírně důležitá, je hrad. Hrady jsou na každé mapě pouze dva, jeden pro Alianci a druhý pro Hordu.

2.2 Jednotky

Jednotky jsou základním stavebním kamenem celé hry, a taky první věc, kterou jsem vymýšlel při jejím vytváření. Každá jednotka má svůj unikátní vzhled. Vytvořil jsem pět druhů jednotek pro obě frakce. Ve hře je tedy deset různých jednotek, ale jen pět z nich mají unikátní atributy. Vedla mě k tomu spravedlnost, aby oba hráči na tom byli stejně. Pokud bych pro každou frakci vymýšlel jinak silné jednotky, bylo by velice těžké to provést tak, aby tyto frakce byly na úplně stejné úrovni, a neměla jedna převahu nad druhou. Nicméně je to jedna z věcí, kterou chci v budoucnu zlepšit, a pokusit se vytvořit více jednotek s různými parametry pro každou frakci, aniž by tím jedna frakce strádala.

Každá jednotka má šest atributů, které ji definují. Útok, obranu, životy, pohyblivost, dosah a cenu. Útok a obrana slouží k soubojům a obléhání hradu. Životy jsou ubírány při soubojích a při klesnutí na nulu jednotka zemře. Pohyblivost značí kolik políček je jednotka schopna přejít za jeden tah. Samozřejmě záleží na typu terénu. Každá jednotka útočí na určitou vzdálenost, která se vztahuje k atributu dosah. Dosah udává přesný počet políček, přes které je možné útok zasadit, nehledě na terén a cenu pole. A nakonec cena



Obrázek 1: Přehled terénů a budov

jednotky udávaná ve zlatě. U všech atributů až na cenu platí, že čím větší hodnota, tím silnější jednotka. Nicméně některé atributy jsou důležitější než jiné.

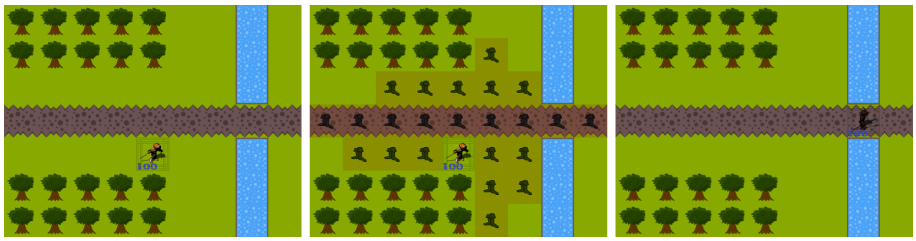
Vzhled	Název	Atributy					
		Útok	Obrana	Životy	Pohyblivost	Dosah	Cena
	Goblin	2	1	100	5	1	120
	Archer	1	1	100	4	2	180
	Werewolf	2	2	100	7	1	250
	Warrior	3	3	100	4	1	300
	Werebear	3	5	100	3	1	400
	Spider	2	1	100	5	1	120
	Mage	1	1	100	4	2	180
	Antilion	2	2	100	7	1	250
	Skeleton	3	3	100	4	1	300
	Minotaur	3	5	100	3	1	400

Obrázek 2: Všechny jednotky s vlastním vzhledem a popisem

2.3 Akce

Pohyb je základní akcí každé jednotky. Během jednoho tahu je možné s jednotkou pohnout pouze jednou. O kolik polí se jednotka bude schopna posunout záleží na jejím atributu *pohyblivost*. Každé pole má určitou cenu, kterou musí jednotka zaplatit, aby se na toto pole mohla přesunout. Tato cena se odráží od typu terénu, kde jít po cestě je rychlejší, než přes trávu. Jednotka se tedy může přesunout o takový počet polí, kde cesta k poslednímu

políčku je menší nebo rovna jeho pohyblivosti. Jednotce můžou také přijít do cesty různé překážky, jako jiné jednotky nebo strom či řeka. Jak takováto akce vypadá v praxi můžeme vidět na obrázku 3. Zvolenou jednotkou je zde *Goblin*, jehož *pohyblivost* je 5. V prvním kroku hráč zvolí možnost pohybu. Poté se mu zvýrazní všechny možné pole, na které se může jednotka přesunout. Když si hráč najede kurzorem nad takovéto pole, a potvrdí akci, je jednotka přesunuta.



Obrázek 3: Pohyb jednotky

Jednotky, které se po mapě pouze pohybují, by byly k ničemu, proto je možné s nimi i útočit. Stejně jako u pohybu, jednotka může za jeden tah zaútočit pouze jednou. Pokud vynecháme výsledek takového útoku, a zaměříme se pouze na jeho provedení, tak nám stačí pouze atribut *dosah*. Tento atribut určuje, jak daleko je jednotka schopna zaútočit. Zde už se nebere v úvahu cena jednotlivých polí, ale každé pole, které stojí v cestě tomuto útoku stojí jeden bod dosahu. Dosah 1 tedy znamená, že jednotka může útočit pouze na sousední jednotky, zatímco jednotka s dosahem 2 může útočit i do jednotek, které jsou vzdáleny o jedno pole dále. Útok probíhá obdobně jako pohyb jak vidíme na obrázku 4, kde je zobrazen útok s jednotkou *Werewolf*, která má dosah 1. Nejprve je u jednotky zvolen útok, a poté jsou zvýrazněny jednotky, na které můžeme útočit. Jelikož má *Werewolf* tak malý dosah, může útočit jen na jednotky *Antilion* a *Spider*, se kterými sousedí. Hráč si zvolí jednotku, na kterou chce útočit, v tomto případě *Spider*, a potvrdí akci. Proběhne útok, ve kterém si obě jednotky uštědří zranění.

Obrázek 4: Útok jednotky s dosahem 1

Na obrázku 5 je vyobrazena tato akce s jednotkou *Archer*, která disponuje dosahem 2,

což ji umožňuje útočit na větší vzdálenosti. Všimněte si, že jednotka s dosahem 2 může dokonce útočit přes vodu, i když je toto pole neobývatelné, a pohybem se přes něj nedá přejít. Taky může útok provést přes ostatní jednotky. V této ukázce útočí *Archer* právě přes řeku na jednotku *Minotaur*. Také by mohl ale zaútočit na jednotku *Skeleton* o dvě pole níž, i když by se zdálo, že by mu v tom měla bránit přátelská jednotka *Warrior*, která je mezi těmito dvěma jednotkami.

Obrázek 5: Útok jednotky s dosahem 2

Už v ukázkách byl vidět i poslední krok této akce, a to vyhodnocení útoku a zranění obou zúčastněných jednotek. Tato část probíhá samovolně po zvolení cíle, není tudíž v režii hráče. V této akci se podle vzorce vyhodnotí atributy obou jednotek, a každé jednotce je uštědřeno zranění odpovídající tomuto vzorci. Poté je útočící jednotka vyčerpána a akce končí. Vzorec pro vypočtení zranění útočníka a obránce je zobrazen ve výpisu kódu 1 . V první části je vypočtena hodnota zranění uštědřená obránci, a v druhé části se počítá totéž pro útočníka. Když se snaží obránce vrátit útok, je také nutné vypočítat, zda je v dostatečné vzdálenosti, aby takový útok provedl. Pokud tedy útočník útočí z větší vzdálenosti, než je obráncův dosah, obránce nevrací žádné zranění.

```
attack_formula = - (((self.attack * self.health) * (2000 /
                    (target.defense * target.health))) / 100)

if distance((self.position), (target.position)) > target.range:
    defend_formula = 0
else:
    defend_formula = - (((target.attack * target.health) *
                        (1000 / (self.defense * self.health))) / 100)
```

Výpis 1: Vzorce pro výpočet uštědřeného zranění. *Self* určuje útočící jednotku a *target* cíl.

Obléhací akce je poslední akcí, kterou může jednotka vykonat. Tato akce slouží k obléhání nepřátelského hradu, kdy se znovu použije vzorec, ve kterém se vyskytuje jen útočný atribut jednotky, a poté je hradu sníženo „zdraví“. Nejsou zde možnosti jako u pohybu nebo útoku, a to z toho důvodu, že soupeřův hrad je jen jeden, a pokud chce jednotka obléhat, není třeba navíc kurzorem hrad zacílit.

2.4 Ovládání

Ovládání je zapotřebí primárně u člověkem ovládaného hráče. Přesto je potřeba ovládání znát, i když jen sledujete tah agenta řízeného počítačem. Pro potřeby demonstrace a také pro přehlednost jsem počítači nastavil při provádění akcí intervaly a k nim i heslovité fráze, které tyto akce reprezentují. Umožňuje to lepší orientaci během tahů. Kdyby byl tah počítače realizován pouhým problesknutím, protihráč by měl problém ihned odhadnout strategii pro tento tah, a stejně by byl nucen zpětně zjišťovat, která jednotka se přemístila, která na kterou zaútočila a podobně. Intervaly jsou u každé akce jinak dlouhé, záleží na tom, jak složitá daná akce je. Například útok je zobrazován pomaleji než pouhý pohyb, protože u útoku je třeba se dívat i na životy jednotek, a na to, kolik si jednotky při souboji uštědřily zranění.

Tímto jsme si zdůvodnili důležitost znalosti ovládání, a můžeme se vrhnout konkrétně na ovládání pro lidského hráče. Ovládacím prvkem je klávesnice, nic víc není potřeba. Pohyb kurzoru po hrací ploše je realizován pomocí šipek. Při stisknutí je kurzor přemístěn o jedno políčko. Šipky lze i podržet pro rychlejší pohyb. Akční klávesou je Enter. Pokud je hráčův kurzor na políčku s jeho jednotkou, tato klávesa aktivuje akční menu. Pomocí tohoto menu se dají jednotce přiřazovat akce. V tomto menu se můžeme pohybovat zase pomocí šipek a Enterem potvrdit akci. Pravý Control reprezentuje klávesu zpět, pomocí níž se hráč může dostat z akčního menu zpátky do hry a nebo vrátit poslední provedenou akci. Mezerník slouží pouze k ukončení tahu, a je možné ho použít jen v základním stavu hry, kdy není žádná jednotka uprostřed akce.

2.5 Uživatelské rozhraní

Obrazovka je tvořena zobrazeným prostředím a spodní lištou, která obsahuje základní informace. Tato lišta je rozdělena na tři části. Levá část zobrazuje zmenšenou mapu s aktuálním rozestavením jednotek. Uprostřed jsou základní informace o hráči a pokud je kurzor na nějaké jednotce, tak také její informace. Pravá část je prázdná do doby, než se hráč rozhodne provést akci. Když se tak stane, nachází se zde výběr akcí. Levá a střední část je vidět po celou dobu hry, tedy i v době, kdy zrovna hraje počítač. V této situaci se jen mění to, že se nezobrazuje pravě výběr akcí. Místo něj je zde již zmiňované heslo reprezentující probíhající akci hráče.

Nad spodní lištou je zobrazeno prostředí, které je podrobně vysvětleno v kapitole 2.1. Samozřejmě není v jeden čas zobrazeno celé prostředí najednou, respektive všechny políčka. Kdyby tomu tak bylo, u velkých map by vznikl problém s políčky, které by musely být velice malé, aby se vtěsnaly do obrazovky. Tudíž je zobrazeno pouze prostředí o sedmnácti políčcích v řadě v devíti sloupcích. Poloha aktuálně zobrazeného prostředí se mění v závislosti na pozici kurzoru. To znamená, že v jednu chvíli jsou zobrazeny pouze políčka, které se vztahují k části mapy, ve které je kurzor. Vzhled celé obrazovky je vyobrazen na obrázku 6.

Obrázek 6: Herní obrazovka

3 Připojení modulu umělé inteligence

Rozhodování inteligentních agentů je založeno převážně na okolních vlivech. Agenti reagují na přátelské i nepřátelské jednotky, a také na prostředí, pozice hradu a podobně. Obsah jednoho tahu jsem rozdělil na dvě důležitá rozhodnutí. Nákup jednotek je prováděn ihned na začátku tahu, a je rozhodnut na základě polohy všech jednotek na mapě. Poté si agent rozdělí jednotky a přiřadí jim zvolené akce, které už jsou spojeny jak s polohami jednotek, tak s prostředím.

3.1 Nákup jednotek

Při nákupu jednotek záleží na aktuálním stavu hry. Inteligentní agent se rozhoduje vždy podle počtu svých jednotek a jednotek soupeře. Také záleží na rozestavení jednotek a na výši zlata, jež daný hráč může utratit. V této oblasti se meze nekladou a je možno naprogramovat nespočetné množství úkonů. Já jsem pro demonstraci implementoval akce pro základní situace. Postupně popíšu tyto akce, jak a kdy mohou nastat, proč se provedou zrovna v dané chvíli, a jak to může situaci pro hráče zlepšit, a naopak soupeři situaci pořádně znepříjemnit.

Ve hře mohou nastat tři základní typy situací, které se odrážejí na tři typy strategií. Negativní - obranná, pozitivní - útočná, a neutrální. Jako pesimista bych rád začal tou negativní, ale pro vysvětlení bude lepší si nejdříve vysvětlit situaci neutrální, která je ve hře nejvíce častá. Tato situace nastává ve chvíli, kdy je hra vyrovnaná. Oba hráči mají v poli své jednotky, snaží se přemoci jeden druhého, ale ani jeden neohrožuje hrad svého protivníka. Tudíž není ani jeden v útočné či obranné fázi. Strategie v této fázi je nejméně agresivní, protože hráč není v tísni, a nemusí okamžitě utrácet všechno zlato, aby nakoupil co nejvíce jednotek. Samozřejmě, že kupuje jednotky podle toho, jak se hra vyvíjí, ale může si i šetřit zlato ke koupi silnějších jednotek. Bohužel, pro inteligentního agenta je tato strategie nejsložitější. Právě proto, že je tak volná, a různorodá, je velice obtížné navrhnout algoritmus, který by efektivně reflektoval situace, které nastanou. V praktické části jsem prozatím agentovi nastavil náhodné nakupování, které protihráč alespoň nemůže předvídat.

Při obranné fázi je hráč v úzkých, protože protihráč ovládá více jednotek v poli, a některé jeho jednotky se dostávají do blízkosti hradu. Agent je nucen spotřebovat všechny své úspory k vyslání oddílu, který bude hrad chránit. Nejdříve se pokusí najmout nejsilnější obranné jednotky, kterými jsou jednotky *Werebear* nebo *Minotaur*, podle aktuální frakce. Tyto jednotky jsou navrhnuty právě na obranu hradu vzhledem k jejím obranným hodnotám. Pokud agent nemá dostatek zlata, zkusí to s jednotkou *Warrior* nebo *Skeleton*. Tyto jednotky už nejsou tak defensivní, ale pořád mají dostatečně silnou obranu, aby zadržely přicházející jednotky. Jestliže je na tom tak špatně, že nemá moc peněz, nezbude mu nic jiného, než si koupit základní jednotku, kterou je buď *Goblin* nebo *Spider*. A pokud nemá vůbec žádné zlato, tak už mu nic nepomůže.

Útočná taktika je v tomto směru mnohem více přímočará. Jelikož jsou všechny agentovy jednotky u nepřátelského hradu, a on je chce podpořit novou vlnou svých jednotek, potřebuje někoho rychlého, kdo by tam dorazil co nejdříve. Proto se k tomu nejlépe hodí

Werewolf nebo *Antilion*. Tyto nejrychlejší jednotky ve hře se dokáží během několika tahů dostat k nepříteli, a ihned útočit. Agent proto utratí všechny peníze za tyto jednotky. Pokud už peníze nemá, tak vyčkává, dokud si nebude moci koupit další.

3.2 Rozhodování a rozdělení akcí

Než hráč začne jednotkám rozdělovat akce, je třeba si rozdělit samotné jednotky. Některé jsou vhodné na útok, jiné na obranu, a jiné zase pro rychlé napadení protivníkových slabších jednotek a ochromení stability. Agent musí uvažovat, jak a proč tyto jednotky rozdělit. Stejně jako u nákupu jednotek, tak i zde platí, že je mnoho možností, jak to provést. Tato práce se těmito možnostmi zabývá pouze teoreticky, protože tato oblast není úplně jejím cílem. Nýbrž je vhodné a do budoucna důležité toto téma neopomíjet.

V praktické části jsem aplikoval jednoduché rozdělení a tím je rozdělení podle aktuální pozice vůči nepřátelskému hradu. Jinými slovy, jednotka, která je nejblíže hradu, začíná svou akci jako první. Určitě to není nejlepší taktika, ale pro demonstraci Pathfindingu je dle mého názoru nejvhodnější. Pokud chci převážně jednotkami pohybovat, je dobré si je seřadit podle pozice. I přesto jsem navrhl a naprogramoval i jiné metody rozdělení. Například rozdělení podle počtu životů, které je také jedno z těch základních.

Když už je pořadí jednotek rozdělené, je třeba přemýšlet nad tím, co chci s danou jednotkou udělat a proč. Každá jednotka má své instrukce, jak se v dané situaci zachovat. Já jsem pro potřeby práce použil klasické skriptování(umělou inteligenci řízenou pravidly), kdy se ptám, zda jsou splněny určité podmínky, a podle výsledku provádím reakce. Nejdřív jednotka zjišťuje, zda jsou v okolí nějaké nepřátelské jednotky. Pokud ano, jednotka okamžitě přejde do útoku, a začne útočit na jednotku, která nejvíce ohrožuje hrad, což jsem prozatím určil jako jednotku, která je k hradu nejblíže. Jestliže se nikde v okolí taková jednotka nenachází, tak se jednotka, s životy rovny alespoň polovině plného zdraví, přesune blíže k nepřátelskému hradu. Jakmile už je dostatečně blízko na obléhání, tak začne hrad obléhat. Pokud je ale jednotka tak zraněná, že má méně než polovinu životů, rozhodne se odpočívat. To znamená, že neprovede žádnou akci, zůstane aktivní, a na konci kola se této jednotce přičte deset životů.

Pravidlo	Stav	Akce
Nákup jednotek		
Silné obranné jednotky	Hrad ohrožen zlato ≥ 400	Kup jednotku Werebear/Minotaur
Střední obranné jednotky	Hrad ohrožen zlato ≥ 300 zlato < 400	Kup jednotku Warrior/Skeleton
Slabé obranné jednotky	Hrad ohrožen zlato ≥ 120 zlato < 300	Kup jednotku Goblin/Spider
Útočné jednotky	Nepřátelský hrad ohrožen zlato ≥ 250	Kup jednotku Werewolf/Antilion
Libovolné jednotky	Neutrální stav hry	Kup náhodnou jednotku
Rozhodnutí akce jednotky		
Útok na hrad	Žádná nepřátelská jednotka poblíž životy ≥ 50	Přesuň se k nepřátelskému hradu a zahaj obléhání
Odpočinek	Žádná nepřátelská jednotka poblíž životy < 50	Odpočiň si, a neprováděj žádné akce
Útok na nepřátelskou jednotku	Nepřátelská jednotka poblíž	Přesuň se k jednotce a zaútoč

Obrázek 7: Tabulka pravidel akcí pro inteligentního agenta

4 Algoritmy umělé inteligence

V této kapitole se budu věnovat dvěma významným odvětvím umělé inteligence, Pathindingu a rozhodování. U každého proberu jejich možné postupy a řešení. Dijkstře budu věnovat nejvíce prostoru, jelikož je to hlavní algoritmus mé práce. Ostatní metody popíšu jen základně, aby si člověk vytvořil představu, jak to asi funguje. Bohužel není cílem této práce podrobně popsat všechny.

4.1 Pathfinding

V každé hře, obsahující nějakou formu prostředí, se nachází postavy, které se v tomto prostředí pohybují. Pohyby postav mohou být náhodné či pevně dané. Je jednoduché implementovat pevně dané cesty, ale jejich výsledkem je dosti neefektivní způsob pohybu, který je možné lehce odhadnout, a ve kterém se postavy mohou i zaseknout. Proto je třeba implementovat algoritmy, které dynamicky vypočítávají efektivní cesty a počítají s překážkami. Tyto algoritmy vypočítávají cestu od počátečního bodu, kde se postava právě nachází, do bodu cílového, kam se chce postava dostat. Většinou to jsou cesty nejrychlejší a nejkratší. A právě tomuto způsobu, který efektivně vyhledává ty nejlepší cesty, se říká Pathfinding.

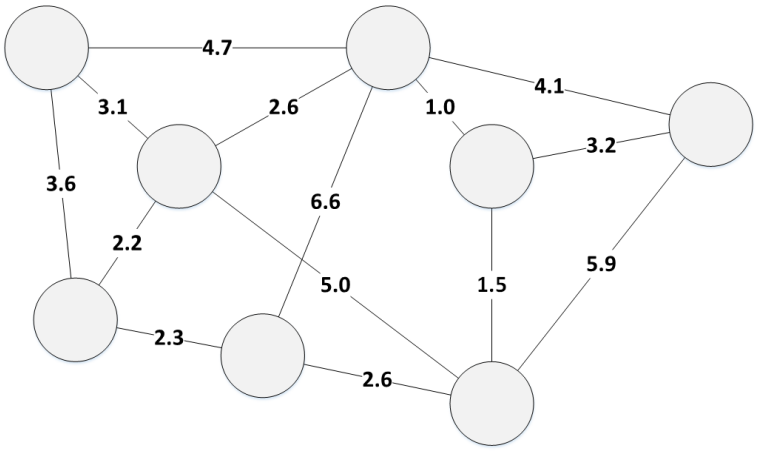
Aby mohl Pathfinding dobře fungovat, je třeba mu předávat informace o prostředí, které by mohly mít vliv na vypočtení nejkratší cesty (již zmiňované překážky). Tyto informace je třeba co nejlépe zjednodušit, a vybrat právě ty důležité, na kterých nejvíce záleží. Pokud by programátor tuto část nezvládl, a předával špatné či méně významné informace, výsledná cesta by pak byla o to méně efektivní. Tyto informace poté programátor zpracuje, a předá je algoritmu ve formě grafu.

Grafem rozumíme uspořádanou dvojici G = (V, E), kde V je neprázdná množina vrcholů a E je množina hran. [4] Každý vrchol(dále definováno jako uzel) reprezentuje prostor, kam může postava dojít, například místnost. Hrany symbolizují přechody mezi těmito prostory. Pokud je možné přejít z kuchyně do obýváku, bude uzel K spojen hranou s uzlem O.

Takovýto graf nám ale ještě nestačí. Abychom mohli porovnávat, která cesta je rychlejší, či kratší, je třeba si hrany ohodnotit. Každý přechod mezi dvěma uzly bude ohodnocen určitým kladným číslem reprezentujícím cenu, kterou je nutné zaplatit, abychom se po této hraně mohli přesunout do dalšího uzlu. Cena je ovlivněna informacemi, o kterých jsem mluvil výše. Pokud je v cestě z kuchyně do obýváku překážka, bude tato cesta náročnější, než kdyby tam ta překážka nebyla. V takovém případě bude ohodnocení této cesty vyšší. Cena může být také ovlivněna vzdáleností a mnoha jinými prvky, které musí programátor brát v úvahu. Takto ohodnocený graf můžeme vidět na obrázku 8.

4.1.1 Dijkstra

Dijkstrův algoritmus je pojmenován po jeho stvořiteli Edsgeru Dijkstrovi. Tento algoritmus je navrhnut pro nalezení nejkratší cesty v grafu, což znamená, že není používán jen v herní průmyslu, ale i v jiných odvětvích. Je také jedním z možných řešení Pathfindingu.



Obrázek 8: Ohodnocený graf

Zatímco Pathfinding vyžaduje pouze nalezení nejkratší cesty z počátečního bodu do bodu koncového, Dijkstra pracuje s počátečním uzlem, a z tohoto uzlu nalézá nejkratší cesty do všech ostatních uzlů. Z toho vyplývá, že nám Dijkstra umožňuje získat více dat, které můžeme použít. Algoritmus sám o sobě dokáže nalézt nejkratší cestu k zadanému uzlu, stejně jak požaduje Pathfinding, ale poskytuje nám navíc více možných cest k danému uzlu, a také cesty k jiným uzlům. Takové informace mohou být použity například při zjištění, že nelze nalézt cesta k uzlu, který chceme. V tomto okamžiku by Pathfinding skončil neúspěchem, a víc by se dělat nedalo. Bylo by třeba najít jiný cílový uzel, a provést celou akci znovu. U Dijkstry bychom mohli zvážit nejkratší cesty do jiných uzlů, a najít cestu, která nás nejrychleji zavede do uzlu nejbližšího. Tuto možnost používám i já, a v další podkapitole jí blíže specifikuji.

Představme si mapu polí jakožto ohodnocený graf z předcházející kapitoly (obrázek 8). Graf obsahuje uzly, které reprezentují políčka na mapě a hrany jako ohodnocené přechody mezi těmito políčky. V tomto grafu si určíme dva uzly jako počáteční a koncový. Řešením bude cesta, jejíž celková cena je minimální cenou všech cest vedoucích z počátečního uzlu do koncového. Těchto nejkratších cest může být více. Pokud bude nalezena více než jedna nejkratší cesta, bude nám v podstatě jedno, která z těchto cest bude řešením. Jednu z nich si vybereme, a ostatní nás nezajímají.

Algoritmus začíná počátečním prvkem, ze kterého se rozdělí po všech hranách na všechny sousední uzly. Při přechodu na další uzel si uchová vzdálenost, kterou už prošel. Při každém rozdělení na nový uzel ověří, zda už tímto uzlem prošel, a pokud ano, tak porovná vzdálenosti. Jestliže je možné se k tomuto uzlu dostat kratší cestou, algoritmus se z tohoto uzlu už nerozšíří a je zde ukončen. Nemá smysl pokračovat v této cestě, když je zřejmé, že nemůže být nejkratší cestou už k žádnému jinému uzlu v grafu.

Dijkstra je prováděna v iteracích, kdy se rozšiřuje z aktuálního uzlu na sousední uzly. Při každém rozšíření si pamatuje informace o své cestě z počátečního uzlu. Jsou to vlastně takové drobečky, které za sebou zanechává, aby při dosažení cílového uzlu mohla určit celou cestu, kterou k tomuto uzlu vykonala. V každém kroku ověřuje daný uzel, na kterém

se nachází. Pokud je prozatímní cesta k tomuto uzlu nejkratší, rozšíří se dále na všechny ostatní uzly, které z tohoto uzlu vedou. Jelikož dopředu nevíme, z kterého uzlu jsme přišli, jdeme do všech sousedních uzlů. To znamená i do toho, ze kterého jsme zrovna přišli. Tento krok je sice zbytečný, ale nezbytný. Nicméně nic se tím nezmění, protože v tomto kroku je ověřován uzel, u kterého je jisté, že už k němu vede kratší cesta (ta, kterou jsme přišli poprvé), tudíž se v tomto kroku algoritmus zastaví, zatímco u ostatních kroků pokračuje dál. V první iteraci, kde aktuálním uzlem je uzel počáteční, je vzdálenost rovna nule, a vzdálenost všech sousedních uzlů je rovna cenám přechodů, které k nim vedou.

Procházené uzly jsou uchovávány ve dvou množinách, nazývaných closed a open listy. V open listu se nacházejí uzly, které už byly při procházení objeveny, ale prozatím vyčkávají na svou vlastní iteraci. Naopak zkontrolované uzly se uchovávají v closed listu. Zbývají nám zatím neobjevené a nenavštívené uzly, které čekají, až budou zařazeny do open listu. Každý uzel se může v jeden čas nacházet pouze v jedné z těchto tří kategorií. Na začátku algoritmu je closed list prázdný, jelikož jsme ještě nezačali procházet grafem, a v open listu je pouze počáteční uzel, kterým je zahájena první iterace. Během každé iterace je z open listu vytažen jeden uzel, který je zpracován, a poté vložen do closed listu. To se děje tak dlouho, dokud nám v open listu zbývá alespoň jeden uzel. Jak už jsem naznačil dříve, je nevyhnutelné, že se k jednomu uzlu dostaneme vícekrát z více možných cest. Pokud se k uzlu dostáváme podruhé, je již uložen v closed listu, a to nás nutí chovat se k němu jinak, než kdyby byl zatím nenavštívený. Stejný problém může nastat, jestliže narazíme na uzel, který se nachází v open listu.

Pokud kontrolujeme uzel, který se nachází v open listu, víme, že tento uzel je kontrolován poprvé, a to znamená, že ještě k němu nevede žádná cesta. Do tohoto uzlu stačí zapsat aktuální vzdálenost a cestu, kterou jsme již prošli. Není třeba řešit nic jiného. Pokud ale narazíme na uzel, který již byl zkontrolován, a vede k němu ohodnocená cesta, nemůžeme tuto cestu jen tak přepsat novou aktuální cestou. Místo toho ověříme, zda je aktuální cesta lepší, než cesta, která už je k tomuto uzlu vytvořena. Pokud ano, smažeme tento uzel z closed listu, a opět ho vložíme do open listu, kde bude poté znovu ověřen, a bude mu připsána aktuální levnější cesta. Pokud je ale aktuální cesta delší, není třeba nic měnit, a uzel zůstává v closed listu.

Výstupem tohoto algoritmu je nejkratší cesta, která vede z počátečního do koncového uzlu. Tu získáme sledováním drobečků, které po sobě zanechávala cesta uložena v koncovém uzlu jako nejkratší. Jakmile dojdeme do počátečního uzlu, tuto cestu obrátíme, protože nechceme cestu z cíle do startu, ale ze startu do cíle. Tímto jsme prošli celým Dijkstrovým algoritmem, doufám, že se vám projížďka líbila. Dost bylo teorie, v další podkapitole předvedu celou Dijkstru v praxi na mnou vytvořeném algoritmu.

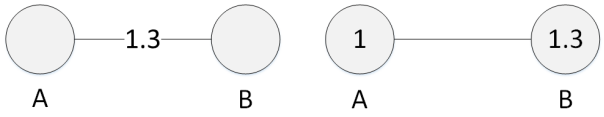
4.1.2 Dijkstra v praxi

Pro jistotu začnu lehkým shrnutím toho, proč používám právě Dijkstru. Pomocí tohoto algoritmu dokážu z jednoho bodu vést cestu do všech ostatních bodů, a zjistit tím, která cesta je do jakého bodu nejkratší. Díky tomu můžu nejen zjistit, kterou cestou se má jednotka vydat, aby došla k nepřátelské jednotce nebo k soupeřově hradu nejrychleji, ale

také na která políčka se může jednotka pohnout v jednom tahu. Každá jednotka má svou pohyblivost, která udává, o kolik polí se může přesunout během jedné akce. Tudíž stačí porovnat vzdálenost ke každému poli s tímto atributem, a všechny pole se vzdáleností menší nebo rovnou tomuto atributu, označit jako ty, na které se jednotka v tomto tahu může přesunout. To pomáhá mimo jiné i uživatelskému rozhraní, kdy lze zobrazit všechna tato pole, a dát tím i lidskému hráči na výběr z polí, na která se může přesunout. Dijkstra tedy není jen pomocníkem pro roboty.

Dále bych chtěl upozornit na změny, které můj algoritmus obsahuje. Graf reprezentující mapu se neskládá z ohodnocených hran, nýbrž z ohodnocených uzlů. To znamená, že když chci přejít na daný uzel, nezjišťuji cenu hrany vedoucí k tomuto uzlu, ale pouze cenu uzlu, na který přecházím. To způsobuje výraznou změnu v samotném chápání prostoru tvořeného mou mapou, a přecházením mezi jednotlivými políčky. Na obrázku 9 máme dva grafy. Levý graf symbolizuje přechod mezi dvěma místy, kde cena přechodu je dána hranou. Pravý graf naopak symbolizuje mnou používaný graf, kde cena přechodu je dána uzlem.

Cena cesty z uzlu A do uzlu B bude na levém grafu rovna hodnotě 1.3. Opačná cesta, tedy cesta z bodu B do bodu A, bude ohodnocena úplně stejnou hodnotou. Nezáleží tedy na směru cesty, vždy bude cesta stejně náročná a dlouhá. Když ale vezmeme v úvahu stejné cesty v grafu pravém, výsledek cesty z bodu A do bodu B nebude roven cestě z bodu B do bodu A. Konkrétně zde je cena cesty do bodu B rovna 1.3, ale cena opačné cesty do bodu A bude rovna hodnotě 1. Je tedy podstatný rozdíl mezi levým a pravým grafem.



Obrázek 9: Rozdíly mezi ohodnocenými uzly a hranami

Z počátku jsem vytvářel algoritmus, který se prováděl rekurzivně. Tento algoritmus zde nebudu více rozebírat, ani prezentovat ukázky kódu, protože v konečné verzi se v této formě nenachází. Né, že by nefungoval. Fungoval tak, jak má, vždy našel nejkratší cestu k cíli. Bohužel ale nebyl tak efektivní, co se týče časové složitosti. U map větších rozměrů (50x30 políček) byl velice pomalý, a rozhodování každé jednotky trvalo poměrně dlouho, řádově i několik minut. Proto jsem ho nahradil efektivnějším algoritmem, který se neprovádí pomocí rekurzí.

Časová složitost Dijkstrova algoritmu je kvadratická. Jak je uváděno v práci [4]: „*Celkový počet kroků Dijkstrova algoritmu potřebný k nalezení nejkratší cesty z výchozího vrcholu u do všech ostatních vrcholů je úměrný zhruba n^2, kde n je počet vrcholů grafu.*“ Složitost je tedy $O(n^2)$. Pokud použijeme výběr uzlu s minimální vzdáleností od uzlu počátečního (Fibonacciho halda), je složitost algoritmu již pouze lineárně logaritmická ($O(n \cdot \log n)$).

Proto jsem místo rekurzí použil pole, do kterého, při procházení, ukládám jednotlivé uzly. Při každé iteraci vyberu z pole jeden uzel, a provedu příkazy, které jsou totožné s použitými v rekurzivní metodě. Už při této změně jsem zaregistroval rychlejší provádění

algoritmu. Nicméně jsem pořád nebyl spokojen, jelikož i více než 5 sekund pro mě nebyl dostatečný výsledek. Řešením v tomto problému pro mě byla právě Fibonacciho halda, která výrazně snížila dobu provádění algoritmu.

Dijkstru jsem rozdělil v podstatě na dvě metody. První metoda přijímá jako argument aktuální uzel a cenu cesty (vzdálenost), která k němu vede, a ověří, zda se může jednotka na daný uzel přesunout. Tento krok je nutný z důvodu různého terénu, které může pole definující uzel obsahovat. Pokud by daným terénem byla voda, je jasné, že se tam jednotka přesunout nemůže, a proto je třeba ověřit, zda tomu tak je, a podle toho postupovat. Pokud proběhne ověření úspěšně, a je možné se k poli dostat, přičte se k vzdálenosti cena aktuálního uzlu. Zde se pak porovnává, zda už neexistuje jiná cesta k tomuto uzlu, která by byla levnější (kratší). Pokud tomu tak je, tento uzel se dále neřeší. V druhém případě je uložena vzdálenost této cesty jako nejkratší, a metoda pokračuje dál s vložením sousedních prvků do open listu. Jelikož je má mapa tvořena čtvercovými políčky, každý uzel, až na ty okrajové, má čtyři sousedy. Tato metoda si ale neuchovává všechny uzly, ze kterých je cesta tvořena. Zde je potřeba druhé metody, která využije ohodnocení cest do všech uzlů k vytvoření nejkratší cesty k uzlu, který si určím. Druhá metoda bude dále popsána podrobněji. Nejprve ale představím a popíšu první zmíněnou metodu, kterou lze vidět celou ve výpisu kódu 2.

```
def find_way(self, mboard, dboard, max_distance):
    open_fields = []
    open_fields.append((0, 0, -1))
    while len(open_fields) > 0:
        field = 0
        for f in range(1, len(open_fields)):
            if open_fields[field][2] > open_fields[f][2]:
                field = f
        min_field = open_fields.pop(field)
        x = min_field[0]
        y = min_field[1]
        distance = min_field[2]
        if distance < max_distance and mboard[x, y] > 0:
            distance += mboard[x, y]
            if dboard[x, y] > distance:
                dboard[x, y] = distance
                open_fields.append((x + 1, y, distance))
                open_fields.append((x - 1, y, distance))
                open_fields.append((x, y + 1, distance))
                open_fields.append((x, y - 1, distance))
```

Výpis 2: Metoda ohodnocující cesty do všech uzlů

Metoda přijímá argumenty *mboard*, *dboard* a *max_distance* (hlavička na obrázku 3). Objekt *mboard* reprezentuje slovník, kde klíčem je pozice políčka a hodnotou cena tohoto políčka. Pokud je například pole o souřadnicích $x = 8, y = 6$ trávou, bude element vypadat následovně: $(8, 6) : 1.3$, kde 1.3 je cena trávního terénu. V případě vody a terénů, které jednotka nemůže navštívit, je cena rovna nule. Dalším argumentem je podobně

fungující objekt *dboard*, který místo ceny pole uchovává cenu momentálně nejkratší cesty z počátečního pole k tomuto poli. Tento objekt vstupuje se všemi hodnotami rovnými 1000, což je mnou vytvořená maximální vzdálenost, které je možno v tomto algoritmu dosáhnout. Je to proto, že při vstupu ještě neznáme žádnou cestu k jakémukoliv poli, takže potřebujeme u každé hodnoty natolik velké číslo, aby ho hned první cesta k tomuto poli nahradila. Zvolil jsem 1000, ale klidně jsem mohl zvolit i 1256 nebo 2560. Výsledek by se nezměnil. Stačí, aby bylo číslo větší, než nejkratší možná cena cesta ke všem polím, a to vše nad 1000 bezpochyby je. Jelikož nepoužívám Dijkstru pouze pro potřeby vyhledávání nejkratší cesty pro inteligentního agenta, ale také ji používám pro zobrazení polí, na které se může jednotka přesunout, je zde nutný i argument s názvem *max_distance*. Pokud chci jen rychle vyhledat všechny možné pole, které může jednotka navštívit, a nechci zbytečně vyhledávat cestu k polím, o kterých vím, že je jednotka během jednoho tahu nenavštíví, je tomuto argumentu přidělena hodnota pohyblivosti dané jednotky. Podle ní algoritmus vždy ukončí momentální cestu, pokud její cena překročí tuto hodnotu. Tohle vše vede k mnohem rychlejšímu provedení algoritmu. Tento argument je zbytečný, pokud naopak chci vyhledat všechny nejkratší cesty do všech polí. Proto je mu v takovém případě přidělena znovu hodnota 1000, což vede k tomu, že tento atribut algoritmus nijak neovlivní.

```
def find_way(self , mboard, dboard, max_distance)
```

Výpis 3: Hlavička metody

V prvních krocích metody inicializuji pole objektů s názvem *open_fields*. Poté jim přiřadím hodnotu prvního uzlu. První dvě hodnoty jsou souřadnice. Tento uzel má souřadnice $x = 0, y = 0$ proto, že je to počáteční uzel. Další hodnota je cena dosavadní cesty k tomuto uzlu. Zvolená konstantní hodnota -1 určuje, že z tohoto uzlu cesta začíná. Asi vám přijde divné, proč zde není nastavena také nulová hodnota. Závisí to na způsobu, jakým ke každé ceně cesty přičítám aktuální cenu uzlu. Pokud bych to prováděl na konci každé iterace, mohla by zde být nula, a algoritmus by fungoval. Jenže já tuto inkrementaci provádím na začátku, a jelikož jsem si při vytváření objektu *mboard* určil, že cena počátečního uzlu bude rovna jedné, v konečném důsledku bude počáteční cena cesty nulová.

```
open_fields = []
open_fields.append((0, 0, -1))
```

V následujícím kódu probíhá již zmiňované zvolení uzlu s nejmenší cenou cesty, která k němu vede. Všechny následující řádky jsou uvnitř cyklu s podmínkou na začátku, tudíž jsou prováděny při každé iteraci tohoto cyklu. Nejdříve je inicializována proměnná *field*, která slouží jako index uzlu, ke kterému vede nejkratší cesta. V dalším kroku se spustí cyklus s řídící proměnnou, který prochází celým polem, a ověřuje, zda obsahuje aktuální uzel kratší cestu. Pokud ano, je do proměnné field uložena hodnota indexu aktuálního uzlu. Pomocí tohoto indexu je poté z *open_fields* vytažen uzel, a vložen do

proměnné *min_field*, kterou budu dále zpracovávat. Informace z uzlu jsou vloženy do tří proměnných, x a y pro souřadnice, a *distance* pro cenu aktuálně nejkratší cesty.

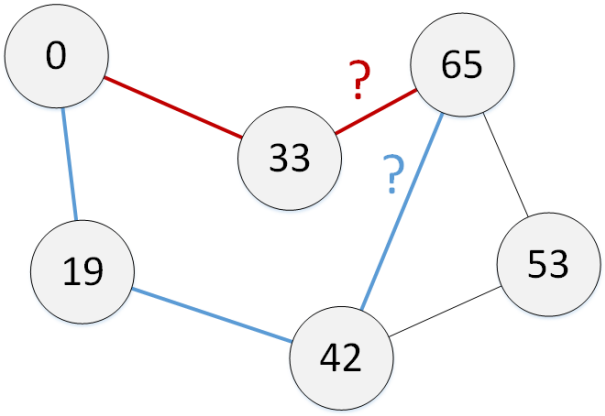
```
while len(open_fields) > 0:
    field = 0
    for f in range(1, len(open_fields)):
        if open_fields[field][2] > open_fields[f][2]:
            field = f
    min_field = open_fields.pop(field)
    x = min_field[0]
    y = min_field[1]
    distance = min_field[2]
```

V posledním kroku se nejdřív dotážeme, zda už jsme nepřekročili maximální cenu, které může cesta nabývat. Přesně tento krok je tím krokem, který odlišuje výsledek algoritmu, jak už jsem uváděl výše. Dále se ověřuje, zda je hodnota *mboard* pro tento uzel vyšší než nula. Pokud by se rovnal nule, znamenalo by to, že pole je neobývatelné, a tudíž na něj jednotka nemůže vstoupit. V tom případě se nic nestane, a iterace se ukončí s tím, že na tento uzel nelze vstoupit, a proto není třeba hledat další cesty. Pokud ale budou splněny obě podmínky, algoritmus pokračuje dál s tím, že je vhodná cesta a vhodný uzel. V dalším kroku je k aktuální ceně cesty připočtena cena tohoto uzlu, jelikož každý pohyb něco stojí. Nyní před námi stojí poslední podmínka, kterou musíme splnit, abychom zajistili, že jde o zatím nejkratší cestu k tomuto poli, a proto je třeba vést z tohoto pole další cesty. Ptám se, zda už k tomuto uzlu existuje kratší nebo stejně dlouhá cesta. Pokud ano, tak není co řešit, a iterace zde končí. Pokud je ale aktuální cesta nejkratší, jsou do *open_fields* vloženy všechny sousední uzly, a je jim předána hodnota této nejkratší cesty.

```
if distance < max_distance and mboard[x, y] > 0:
    distance += mboard[x, y]
    if dboard[x, y] > distance:
        dboard[x, y] = distance
        open_fields.append((x + 1, y, distance))
        open_fields.append((x - 1, y, distance))
        open_fields.append((x, y + 1, distance))
        open_fields.append((x, y - 1, distance))
```

Tak a to byl poslední krok celé iterace. Tento proces se opakuje pořád dokola, dokud se v *open_fields* nachází uzel, který musí být ověřen. Pokud už v tomto objektu nic není, je celá metoda ukončena, a v objektu *dboard* je u každé pozice uložena cena nejkratší cesty.

Nyní máme všechny uzly ohodnocené nejkratší možnou cenou, kterou musíme ujít, abychom se k tomuto uzlu dostali. Jelikož je to vše, co máme, nevíme, kudy cesta do těchto uzlů vede. Ještě jsme neposbírali drobečky, které nám tato nejkratší cesta zanechala. Mějme graf na obrázku 10. Čísla uzlů definují cenu nejkratší cesty k tomuto uzlu. Víme, že abychom se dostali k uzlu 65, je třeba mít 65 bodů pohybu. Bohužel zatím ale nevíme, kudy bychom se měli vydat. Musíme posbírat drobky, a to uděláme nyní. Budeme tyto drobky následovat, a tím zjistíme, která cesta je ta pravá.



Obrázek 10: Ohodnocený graf

V této fázi máme tedy výše zmíněný graf, který jsme získali provedením první metody. Nyní je třeba předat tento graf metodě druhé, a vypočítat přesnou strukturu nejkratší cesty. Jelikož je tato metoda mnohem složitější, a využívá mnoho dalších metod, které jsou neméně složité, nebudu jí popisovat celou, a pokusím se jen na některých částech názorně předvést jednotlivé kroky, které provádí.

Tato metoda je rekurzivní, kdy každá iterace slouží k procházení jednoho uzlu, tak jak u prvního algoritmu. Argumentem metody je pouze aktuální uzel, kde se při první iteraci nastavuje uzel počáteční. V první části metody je ověření, zda aktuální uzel již náhodou není uzlem cílovým. Pokud ano, je tento uzel uložen do proměnné *last_node* a metoda tímto končí.

```
def step(self, actual):
    if actual == self.target_node:
        self.last_node = actual
        return
```

Jestliže je ale tento uzel jen dalším z uzlů, kterými můžeme vést cestu, jsou jeho sousední uzly uloženy do open listu, kde čekají na další zpracování. Do proměnných x a y je vložena pozice aktuálního uzlu. Před každým sousedním uzlem se ptám, zda tento uzel existuje. Toto ověření je nutné u krajních polí, které nemají všechny čtyři sousedy. Například políčko, které se nachází v levém horním rohu má jen dva sousedy, a to pravého a dolního. Pokud daný uzel existuje, je pomocí metody *add_node* vložen do open listu. Metoda *add_node* mimo jiné ověřuje, zda se už tento uzel v open listu nenachází.

Následuje odstranění aktuálního uzlu z open listu a jeho následné vložení do closed listu. To se děje v metodě *del_node*. Dále se ptáme, zda je ještě v open listu nějaká položka. Pokud ano, je z tohoto listu vytažena položka s nejkratší vzdáleností. Jestliže se již v open listu nenachází žádný uzel, znamená to, že neexistuje cesta z počátečního uzlu do uzlu koncového. To se může stát, pokud v cestě stojí překážky, jako například voda nebo jiná jednotka, přes kterou nemůže jednotka přejít. V takovém případě nemůžeme dosáhnout

```
x, y = actual.position
if x > 0:
    self.add_node(self.nodes[tup(actual.position, '-', (1, 0))], actual)
if x < board.width - 1:
    self.add_node(self.nodes[tup(actual.position, '+', (1, 0))], actual)
if y > 0:
    self.add_node(self.nodes[tup(actual.position, '-', (0, 1))], actual)
if y < board.height - 1:
    self.add_node(self.nodes[tup(actual.position, '+', (0, 1))], actual)
```

vysněného cíle a je třeba se spokojit s uzlem, na který se můžeme přesunout, a který se v nejlepším případě nachází blízko našeho cíle. Tímto tato metoda končí a do proměnné *last_node* je uložen cílový uzel.

```
self.del_node(actual)
if len(self.open_list) > 0:
    node = self.open_list[0]
    for n in self.open_list:
        if n.distance < node.distance:
            node = n
else:
    node = self.closed_list[0]
    for n in self.closed_list:
        if distance(n.position, self.target.position) < \
                distance(node.position, self.target.position):
            node = n
    self.last_node = node
    return
self.step(node)
```

Cílový uzel je dále zpracován. Jak vidíme ve výpisu kódu 4 do proměnné *node* je uložena instance tohoto uzlu. V cyklu s podmínkou na začátku je postupně procházená cesta uzel po uzlu, a každý tento uzel je uložen do pole s názvem *move_fields*. Jednoduše následujeme drobky, a ty sbíráme. Poslední příkaz převrátí danou cestu, aby začínala v počátečním bodě, a tuto cestu vrátí jako výsledek metody.

```
node = self.last_node
while node.has_parent():
    move_fields.append(node.position)
    node = node.parent
return move_fields[::-1]
```

Výpis 4: Získávání celé cesty z počátečního do koncového uzlu a její následné převrácení

4.2 Rozhodování

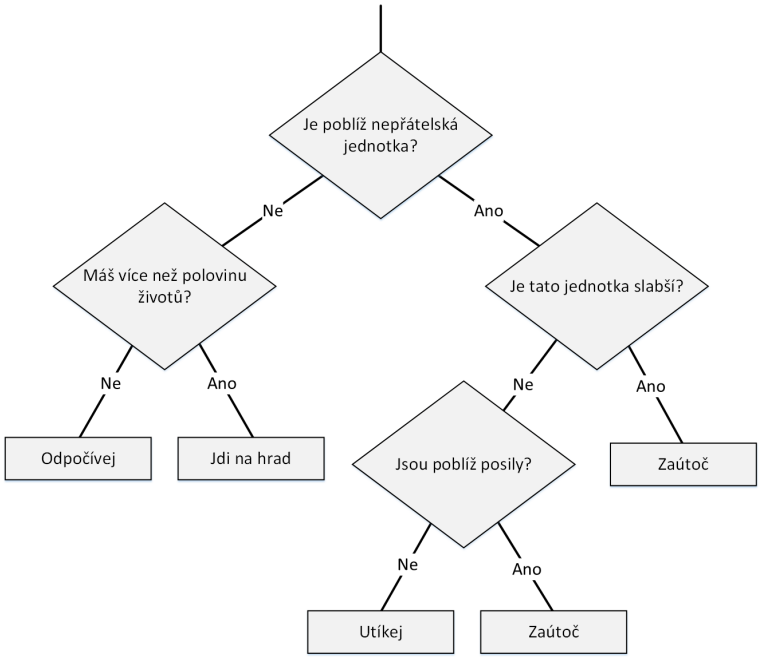
Rozhodování je jedna z prvních věcí, která nás napadne, když se řekne umělá inteligence. Umění správně se rozhodnout, co v dané situaci provést, je velice důležité. Přesto je tato část jen malým krůčkem k vytvoření efektivního inteligentního agenta. Nejprve je třeba získat všechny možné informace, díky kterým získá agent vědomosti, které bude moci dříve či později použít pro správné rozhodnutí. Tyto vědomosti se rozdělují na vnitřní a vnější. Vnější vědomosti určují povědomí agenta o okolním prostředí, například kolik jednotek je poblíž, jak jsou silné, kolik mají životů a podobně. Vnitřní vědomosti určují informace o samotném agentu, jeho zdraví, jeho síla, jeho cíl a tak dále. Každá akce ovlivňuje tyto znalosti určitým způsobem. Některé akce dodají pouze interní vědomost, jiné pouze externí, a některé oboje. Útok je jedna z akcí, která ovlivňuje obě tyto vědomosti. Když agent zaútočí na nepřátelskou jednotku, získá novou informaci o tom, že jednotce bylo uštědřeno zranění, a její počet životů klesl, také ale dostane informaci o tom, že i jeho počet životů klesl, a že je tímto útokem vyčerpán. Interní vědomosti jsou podstatně méně nápadnými a je těžší je identifikovat a poté zpracovat. Přemístění jednotky, nebo jiná změna na mapě je zřejmá okamžitě, ale určení cíle jednotky podle toho, kam má namířeno, a podle jeho minulých tahů, je mnohem složitější. Přesto jsou tyto vědomosti hodně důležité, a je třeba je brát v úvahu stejně tak jako vědomosti externí. Množství vědomostí a jejich důležitost záleží na samotné hře. V některých hrách stačí agentovi vědět pár informací, aby se dokázal rozhodnout, v jiných je naopak těchto informací tolik, že jejich důležitost je velice významná.

4.2.1 Rozhodovací stromy

V kapitole 1.2.1 jsme si řekli něco málo o skriptování. Teď můžeme pokračovat, protože rozhodovací stromy jsou právě jedním z příkladů této technologie. Tak jako skriptování i rozhodovací stromy jsou jednoduché, jsou rychlé, a je snadné je pochopit i implementovat. Principem je sběr vědomostí, dle kterých určujeme akci, která se má provést, a aplikování těchto vědomostí do rozhodovacího stromu. Stromem zde rozumíme souvislý graf, který neobsahuje cyklus. [4] Je tvořen mnoha rozhodnutími, kterými postupně procházíme na základě našich vědomostí. Rozhodnutí provádíme dokud nedojdeme k listům tohoto stromu, kde už není třeba se rozhodovat, ale rovnou provést akci, která je v tomto posledním kroku obsažena. Většina stromů obsahuje jednoduché rozhodování s dvěma možnými odpověďmi, většinou True nebo False.

Samotná rozhodnutí jsou velice jednoduchá, jelikož převážně kontrolují pouze jedinou informaci. Záleží pouze na způsobu implementace a na datových typech, které jsou při získávání informací používány. Například je velký rozdíl, zda agent dostane pouze informaci, že je nějaká jednotka v okolí (True nebo False), nebo získá i polohu této jednotky. Při rozhodování je také možné používat metody, které pomohou řešit více sofistikované problémy. Samotná metoda sice může vracet pouze boolean hodnotu, ale uvnitř metody může být více informací, které jsou zpracovávány. Žádné jedno rozhodnutí ale nepoužívá větvení informací a nevztahuje se na více informací zároveň. Jinými slovy, v tomto stromě se nemohu zeptat „Jsou poblíž nějaké jednotky a mám více než 50 životů?“. Pokud se

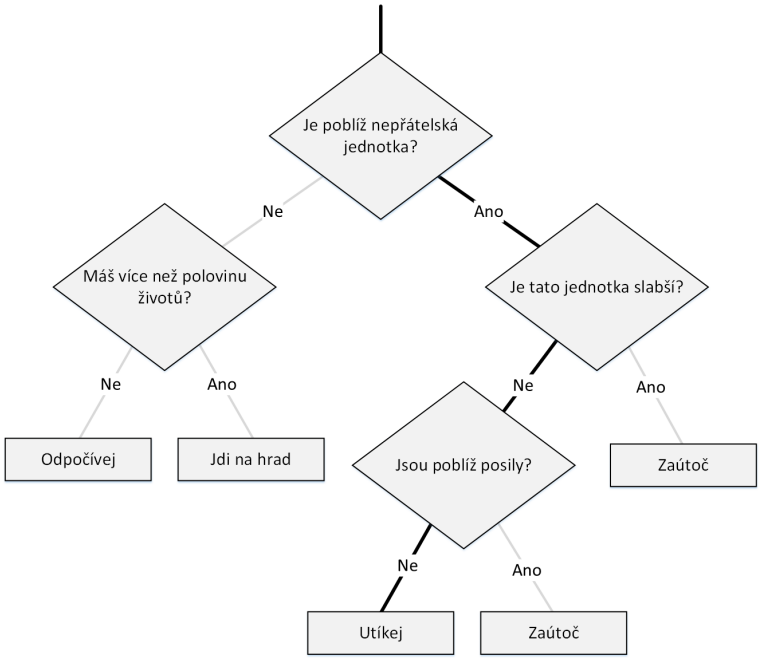
chci takto zeptat, musím rozhodnutí zařadit za sebe a rozdělit je na dva uzly. Nejdříve se zeptám, pokud jsou poblíž nějaké jednotky, a pokud ano, tak přejdu na rozhodnutí, kde se ptám, zda mám více než polovinu životů. Navíc může být každá akce obsažena na více místech, a může k ní vést jiné rozhodnutí. Ukázku takového stromu můžete vidět na obrázku 11.



Obrázek 11: Příklad rozhodovacího stromu

Představme si situaci, kdy je poblíž jednotky tvořící rozhodnutí, nějaká nepřátelská jednotka s větším útokem a více životy. V tom případě jednotka projde stromem, a nakonec se zeptá, zda jsou poblíž nějaké posily ve formě přátelských jednotek. Pokud ne, tak jednotka musí utéct, aby se nestala potravou pro silnější nepřátelskou jednotku. Následující strom (obrázek 12) demonstruje tuto situaci na tom samém rozhodovacím stromě.

Výpis kódu 5 zobrazuje mou rozhodovací metodu pro zvolení akce jednotky. Tato metoda je tvořena rozhodovacím stromem, kde postupně ověřuji vědomosti jednotky, a určuji ji akci, kterou vykoná. V prvním kroku si do pole *enemies* vložím instance všech jednotek, které jsou poblíž. Pokud se délka tohoto pole rovná nule, nenachází se poblíž žádná jednotka. V tom případě je třeba zjistit, zda se jednotka cítí dostatečně zdravá na to, aby mohla jít vstříc nepřátelskému hradu. Pokud ano (má více než 50 životů) namíří si to k hradu pomocí metody *move_to*. Výstup této metody je odpověď na otázku, zda se již jednotka nachází hned u nepřátelského hradu. Pokud ano, jednotka začne hrad obléhat. Jestliže je jednotka natolik slabá, že nemůže jít na hrad, zvolí si odpočinek, který není definován žádnou akcí. Metoda *show* pouze zobrazí heslovitou frázi, aby hráč věděl, která akce byla vykonána. Naopak, pokud se to v okolí jednotky hemží nepřátelskými jednotkami, zvolí si jednotka první cíl z pole *enemies*. Jelikož jsou jednotky v tomto poli



Obrázek 12: Rozhodovací strom aplikovaný na danou situaci

seřazeny podle vzdálenosti od hradu, první jednotka je právě ta, která hrad ohrožuje nejvíce. Jakmile se jednotka dostane k této jednotce dost blízko, zaútočí.

```
enemies = self.near_enemies(self.get_action_board(unit.position, unit.movement * 2))
if len(enemies) == 0:
    if unit.health > 50:
        if self.move_to(unit, self.enemy_castle):
            self.siege(unit)
    else:
        self.show(unit.position, 'REST!', 6)
else:
    if self.move_to(unit, enemies[0]):
        self.attack(unit, enemies[0])
```

Výpis 5: Rozhodovací metoda

4.2.2 Stavové automaty

V mnoha případech agenti stále dokola opakují tu jistou akci, dokud nedojde ke změně jejich vědomostí, a neobjeví se faktor, na který budou reagovat jinak. Například bojovník ve hře *Halo* zůstává na svém stanovišti, dokud se do jeho zorného úhlu nepřiblíží hráč. V takovém případě změní své postavení, a začne útočit. Tento typ rozhodování se používá v každé herní umělé inteligenci. Vždy agent vykonává určitou akci, dokud se

něco nezmění. A jelikož ve hře většinou mění okolí hráč, je to právě on, na koho poté agent reaguje, a kdo ho donutí změnit své chování. Tyto akce a rozhodnutí by bylo možné implementovat s použitím rozhodovacího stromu. Nicméně je efektivnější použít stavové automaty, techniku, která byla pro tento způsob rozhodování stvořena.

Práce [3] definuje stavový automat jako pětici A = $(Q, \sum, \delta, q_0, F)$, kde

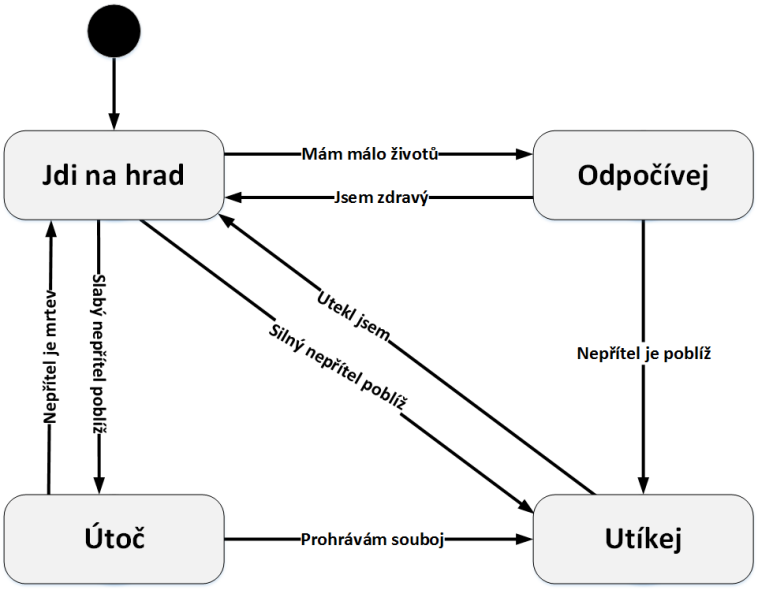
- Q je konečná neprázdná množina stavů,
- Σ je konečná neprázdná množina zvaná abeceda,
- $\delta : Q \times \Sigma \to Q$ je přechodová funkce,
- $q_0 \in Q$ je počáteční stav,
- $F \subseteq Q$ je množina přijímacích stavů

Automat se skládá ze stavů a přechodů. Každý stav reprezentuje určité akce a chování, které jsou vykonávány agentem nacházejícím se na tomto stavu. Každý agent tyto akce vykonává, dokud se něco nezmění, a on není nucen stav opustit, a přesunout se na stav jiný. K opouštění a přesouvání se mezi stavy slouží přechody, které jsou definovány změnou, která nastala. Každý přechod se nachází pouze mezi dvěma stavy, které spojuje, a má jen jeden směr přechodu. Kdykoliv se ve hře změní situace, a agent je na stavu, ze kterého vede přechod do stavu jiného, tak ověřuje, zda jsou splněny podmínky přechodu. Pokud ano, přesune se do jiného stavu, pokud ne, setrvá tam, kde je. Obrázek 13 zobrazuje jednoduchý automat, který je svou situací podobný předchozí ukázce stromů. Černý kruh definuje stav, na který se agent přesune při zahájení hry. V tomto případě hned po zapnutí hry jde jednotka na soupeřův hrad.

4.2.3 Minimax

Tento algoritmus je užívaný v tahových hrách, kde se střídají dva hráči provádějící své tahy, a slouží k určování nejlepšího tahu, který by měl počítačem ovládaný hráč vykonat. Minimax [5] pracuje na základě prohledávání stavového prostoru, kde bere v úvahu všechny možné akce, které může hráč během svého tahu vykonat. Každý hráč má mnoho možností, jak svůj tah provést. Může zaútočit s jednou jednotkou, a s druhou se pouze přesunout nebo naopak a podobně. Všechny tyto možnosti jsou započítávány do množiny všech možných stavů, které mohou nastat. Poté jsou vyhledávány všechny možné stavy, které mohou být reakcí na každý z těchto stavů. Toto se děje stále dokola, dokud programátor neurčí hloubku celého prohledávání. Například si může zvolit, že bude algoritmus uvažovat pět tahů dopředu.

U deskových tahových her je jedna velká výhoda, díky které tato technologie může fungovat. Tou výhodou je, že známe, nebo můžeme určit konečný počet všech možných akcí, které může hráč při svém tahu vykonat. A pokud známe všechny možné výsledné stavy tohoto jednoho tahu, můžeme také určit všechny možné stavy, které mohou nastat během tahu příštího. A takto můžeme iterovat, dokud nedojdeme k poslednímu tahu, ve kterém může nastat jen stav, kdy hráč buď prohraje nebo naopak vyhraje. Konečný stav, na



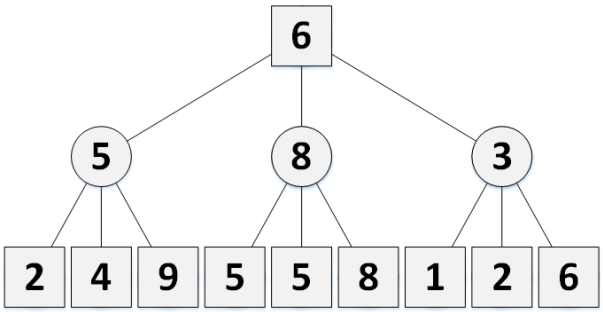
Obrázek 13: Příklad automatu

který není možné reagovat, bude listem stromu, kde každý uzel reprezentuje jeden možný tah. Minimax využívá znalostí těchto stavů, a snaží se najít v takto vytvořeném stromě tu nejlepší větev, která vede k vítězství. Aby mohl strom tímto způsobem prohledávat, je nutné každý stav ohodnotit na základě toho, jak moc prospěšný tento stav hry pro hráče je. K ohodnocení slouží statická hodnotící funkce, kterou navrhuje programátor.

Účel této funkce je ohodnotit předaný stav z pohledu jednoho hráče, a zjistit, zda je pro hráče tato situace výhodná či nikoliv. Pokud funkce vyhodnotí stav kladně, je hráč ve výhodě oproti svému soupeři, a pokud záporně, je to právě naopak. Remíza nastane, pokud je ohodnocení nulové. Výstup většinou nabývá celočíselných hodnot, které se rychleji porovnávají než například čísla s plovoucí desetinnou čárkou. Implementace této funkce je ale čistě na programátorovi. Rozmezí hodnot, kterých může každý stav nabývat, také není přesně dané. Díky této funkci si může agent zvolit ten nejlepší tah, reprezentovaný nejvyšším číslem, na základě ohodnocení všech dostupných tahů, které může vykonat. Zní to jednoduše, ale vytvořit perfektní statickou hodnotící funkci je velice obtížné. Člověk umí, při pohledu na stav hry, určit, zda prohrává či vyhrává, a zda jeho tah tuto situaci změní. Umělá inteligence postrádá toto privilegium, proto je třeba zkoumat více tahů dopředu. Předpokládat, jak bude na jeho tah reagovat soupeř, a vypočítat další reakci na tu jeho.

Nyní už víme, že pro hráče je nejdůležitější vybrat si tah s nejvyšším ohodnocením. Soupeř se naopak snaží hráče dostat do co nejhorší pozice, tudíž si vybírá ohodnocení nejnižší. Při procházení tahů, a střídání hráčů je třeba mít na paměti, že jednou chce hráč ohodnocení nejvyšší a jednou zase nejnižší. Střídání probíhá při každém vnoření, dokud nedojdeme k poslednímu tahu. Takovéto střídání mezi nejvyšším a nejnižším ohodnocením se nazývá minimaxing.

Herní strom na obrázku 14 obsahuje všechny stavy hry během dvou tahů. Hráč začíná tah ze stavu ohodnoceném číslem 6, což znamená, že je momentálně ve výhodě, a je na tom lépe než jeho soupeř. Když se hráč podívá na listy tohoto stromu, zjistí, že může skončit na čísle 9, což by pro něj byl určitě dobrý výsledek, vzhledem k tomu, že nyní je na čísle 6. Nicméně hráč musí předpokládat, že mu protihráč nedovolí dostat se do tak dobré pozice, a naopak se bude snažit o co nejmenší ohodnocení posledního tahu. Proto by pro něj první větev neskončila číslem 9, ale číslem 2, což by nakonec nebyl dobrý výsledek. Při druhé větvi by nemohl hráč dosáhnout tak vysokého ohodnocení, ale nejhůř na tom může být s číslem 5, ať už protihráč udělá cokoliv. Tudíž je pro hráče prozatím lepší posunout hru tímto směrem. Třetí větev je už od pohledu ta nejméně lákavá. Nejvyšší číslo je zde 6, což je stav, na kterém je hráč nyní, a nejmenší je číslo 1, nejhorší možný výsledek, kterého může dosáhnout. Správným řešením tedy bude, že hráč provede akce, které ho dostanou do stavu s číslem 8. A poté může doufat, ať je protihráč natolik hloupý, že provede tah, který končí opět číslem 8.



Obrázek 14: Minimax

5 Diskuze nad problémy a návrhy do budoucna

Nejdříve bych definoval slovo problémy. Tedy samozřejmě v kontextu, v jakém ho budu tady používat, nebudu se fušovat do řemesla lingvistům, na to nemám dostatečné znalosti. Problémem není, že by výsledná hra nefungovala a nedala se hrát. Hra je v pořádku, hrát se dá jak mezi dvěma lidmi, tak i s počítačem. Jako problém zde budu uvádět nedostatky umělé inteligence obsažené v agentovi. V průběhu celé práce jsem upozorňoval na mnoho problémů, kterým jsem se dostatečně nevěnoval z důvodu nedůležitosti v této práci. Těmto problémům se budu věnovat později, a proto je více rozeberu v příštích řádcích.

Prvním problémem je nevyužitý potenciál strategií, které jsem použil. Rád bych vytvořil více komplexní strategie, které by mnohonásobně zvýšily nepředvídatelnost agenta a tím i jeho obtížnost. Proto zvýším počet informací, které zahrnu do jeho vědomostí, a podle kterých se bude rozhodovat. Tento problém se dá rozšiřovat mnoha způsoby. Já budu rozšiřování zakládat hlavně na praktické zkušenosti ze hry. Myslím si, že to je nejlepší způsob, jak odhalit nedostatky, než jen teoreticky hádat, které rozhodnutí je nejlepší.

Dalším problémem je celková omezenost akcí. Akce ve hře jsou opravdu jen základní, a po chvíli hraní začíná být hra až moc stereotypní. Člověk už tam nemá co objevit, či jak vymýšlet nové strategie. Rád bych vedle akcí hlavních přidal více akcí vedlejších. Tyto akce by nebyly tak nezbytné pro samotnou hru, ale přidávaly by více možností. Jelikož je hra tahovou strategií, je malá pravděpodobnost, že by mnoho akcí hru akorát pokazilo. Přesto bych chtěl přidat zatím jen asi dvě akce navíc, které by se týkaly výhradně okolí. Interakci s budovami a také interakci s přátelskými jednotkami. Interakce s budovami by mohla zahrnovat léčení nebo úkryt před nepřáteli a interakce s přátelskými jednotky by mohla působit na morálku jednotek, a zvyšovat jim tím sílu.

Více hráčů. Tento krok jsem zamýšlel už od prvního pokusu tuto hru vytvořit. Při hře lidských hráčů by nebyl problém aplikovat pravidla pro čtyři hráče, kteří by mezi sebou zápasili. Ale pozměnit pravidla inteligentního agenta by už bylo mnohem složitější. Je třeba brát v úvahu mnohem více faktorů. Myslím si, že je lepší nejdřív dovést hru dvou hráčů k lepším výsledkům, a až pak zamýšlet expanzi. Přesto je to jedna z věcí, které chci určitě v budoucnu vytvořit. A nezůstane to jen u počtu hráčů, ale také bych chtěl přidat různé herní módy. Mód „všichni proti všem“ by byl samozřejmý, přidal bych i týmový mód, kde by si hráči mohli vybrat týmy a hrát tak třeba dva proti dvěma. Dalším módem by byla hra, kde by se více hráčů muselo spojit proti jednomu počítačem ovládanému soupeři. Ten by měl obrovskou výhodu nejen v základním množství zlata, ale také v najímání jednotek, ke kterým ostatní hráči nemají přístup.

„Více terénu, více terénu, více terénu!“ toto jsem si říkal většinu doby, během které jsem hru implementoval, nebo když jsem vymýšlel nové mapy. Toho terénu je tam opravdu málo. Jednotky se mohou pohybovat pouze po dvou typech, trávě a cestě. Chtěl bych přidat bažiny, které budou těžší na procházení. Také bych chtěl přidat terénům více vlastností, a výrazně tak ovlivnit Pathfinding. Například kopce a hory by navíc přidávaly bonusovou obranu, a odpočívadla by mohly přidávat na konci kola životy jednotkám, které je navštíví. Věže, které už jsou součástí prostředí, ale jsou jen na okrasu, by byly středem pozornosti obou hráčů, protože by na začátku každého tahu vlastníkovi přidávaly

bonusové zlato. Už by se nebojovalo jen o hrady, ale i o kontrolu nad těmito věžemi, které by poskytovaly vlastníkovi ohromnou výhodu. Kontrola by spočívala v obsazení jednotky daného políčka. Nebo zavedení nové akce obsazení, kde by bylo možné věže obsadit, a vlastnit je, dokud se je nepokusí obsadit protihráč.

Počet jednotek by měl narůstat. Postupně bych chtěl přidávat nové jednotky, a taky přidat některým jednotkám jedinečné schopnosti. Prozatím jsou jednotky definovány šesti atributy, které se různě mění v závislosti na síle jednotky. To bych chtěl změnit. Například vytvoření jednotky, která je zdánlivě slabá v poměru atributy/cena, ale má schopnost oživovat padlé. Nebo jednotka, která dodává morálku ostatním jednotkám a tím zvyšuje jejich útočný atribut. Také bych se chtěl postupně vrhnout do problému, který jsem již uváděl. Vytvořit pro každou frakci jedinečné jednotky, které budou tuto frakci definovat. Místo Aliance a Hordy bych navrhl jiné, jedinečné frakce. Například Peklo, kde by se dali najímat rohatí pekelníci a plamenní oři. Hvozd, který by obývali elfové a prastaří enti.

Nakonec přijdou na řadu méně důležité vylepšení, kterým zatím nebudu věnovat tolik pozornosti, a které jsou jednodušší k vytvoření. S novým terénem určitě přibudou nové mapy. Také bych chtěl pro mou hru ucelenější grafiku. Momentálně hra obsahuje mnoho obrázků, které jsem získal z různých zdrojů, protože sám nejsem schopen docílit v grafice žádných výsledků.

6 Závěr

Cílem této práce bylo seznámit se s problematikou umělé inteligence ve hrách a navrhnout inteligentního agenta pro tahovou strategii. Avšak tato problematika je velmi rozsáhlá, tudíž není v mezích této práce, abych se věnoval všem jejím aspektům. Zvýšenou pozornost jsem věnoval algoritmům pro vyhledávání nejkratší cesty. Tyto algoritmy jsem si podrobně prostudoval, pochopil je, a poté je aplikoval v mnou již vytvořeném prostředí fantasy tahové strategie. Navíc jsem se pokusil věnovat i algoritmům pro rozhodování, které jsem si rovněž prostudoval. Výsledná hra je hratelná jak z pozice dvou hráčů, tak i z pozice hráče proti počítači a počítače proti počítači. Rozhodování inteligentního agenta je v rámci možností na dobré úrovni, a schopnost vyhledat pro jednotky nejkratší cestu k cíli je velice efektivní. Agent vyhledá nejkratší cestu ve velmi rychlém čase.

7 Přílohy

Příloha A

Uživatelská příručka je v elektronické podobě uložena jako *prirucka.txt* na přiloženém DVD.

Příloha B

Třídní diagram je uložen jako *diagram.png* na přiloženém DVD.

Příloha C

Zdrojové kódy programu jsou uloženy v adresáři *program* na přiloženém DVD.

8 Literatura

[1] MILLINGTON, Ian a John David FUNGE. *Artificial intelligence for games*. 2nd ed. Burlington, MA: Morgan Kaufmann/Elsevier, c2009, xxiii, 870 p. ISBN 01-237-4731-7.

[2] *Giant Bomb* [online]. Dostupné z: http://www.giantbomb.com/rubber-band-ai/3015-35/

[3] JANČAR, Petr. *Úvod do teoretické informatiky – učební text*. první. 2007.

[4] KOVÁŘ, Petr. *Úvod do Teorie grafů – učební text*. 2012.

[5] *Flying Machine Studios* [online]. Dostupné z: http://www.flyingmachinestudios.com/programming/minimax/